

ALGORITHM FOR LOCALIZATION OF A JAVA APPLICATION USING REFLECTION API AND A CUSTOM CLASS LOADER

1 **Technical Field**

2 The technical field relates to JAVA® applications, and, in particular, to JAVA®
3 applications whose user interfaces support multiple human languages. (JAVA is a
4 trademark of Sun Microsystems, Inc.)

5 **Background**

6 Localization is a process of adding support of another human language (referred
7 to as target language) to an application. Localization can be divided into two parts:
8 localization of user interface data and localization of other data. User interface data
9 includes a limited number of data elements, which are defined during application
10 development. User interface data elements can be defined in text, graphics or other form.

11 User interfaces of many applications can be localized if an environment supports
12 the target language. The term “environment” includes operating system, programming
13 language and application libraries, fonts and character tables. If the environment supports
14 the target language, the localized version of the application for the target language can be
15 created using the following “straightforward” method: take source or binary code of the
16 application and replace string constants with corresponding string constants in the target
17 language. The straightforward method is expensive for large applications because: a) the
18 localization process must be repeated for every new version of the application; and b) the
19 personnel that perform the localization must deal with the application source or binary
20 code.

21 A widely known solution to the problems of the straightforward method is to keep
22 language-specific data in resource files, because multi-language support in application
23 architecture can reduce cost of localization. The resource files contain key-value pairs,
24 and the application locates data elements in the resource files by finding a value
25 corresponding to a certain key. However, this approach also has a disadvantage: the
26 programmer has to write a function call for every localized string, which results in large
27 amount of function code dedicated to localization, for example, to retrieve localized
28 strings for a current language.

29 The Java programming language simplifies the implementation of resource file-
30 based solution by providing `java.util.ResourceBundle` class in the standard
31 library. This class provides support for human language-specific resource files.

1 However, if the programmer uses class `ResourceBundle` directly, the amount of code
2 dedicated to localization still remains large. In addition, localization-specific code is
3 typically spread all over the application code, making it difficult to read and maintain.

4 **Summary**

5 A method for localization of a Java application includes locating a plurality of
6 localizable variables of a class using a custom class loader, finding a corresponding
7 resource file for a current language for each localizable variable, and calculating a key for
8 each localizable variable. The method further includes finding a localized string in the
9 resource file corresponding to each key, and assigning the localized string to the
10 corresponding localizable variable of the class. Accordingly, the custom class loader
11 provides localization of the class during class loading.

12 The method and corresponding apparatus for localization reduce complexity of
13 the Java application by eliminating the function code dedicated to localization. In
14 addition, the method and apparatus for localization increase productivity of engineers
15 who write language-independent code, and reduce memory consumption of classes that
16 use localized strings. Since all localization is accomplished when the class is loaded and
17 since the code that uses localization is independent of the code performing localization,
18 the method and apparatus for localization ensure better performance of the application
19 and afford better code reusability.

20 **Description of the Drawings**

21 The preferred embodiments of a method and apparatus for localization using a
22 reflection application programming interface (API) and a custom class loader will be
23 described in detail with reference to the following figures, in which like numerals refer to
24 like elements, and wherein:

25 Figure 1 illustrates a standard class loading process;

26 Figure 2 illustrates a custom class loading process that implements an exemplary
27 method for localization using a reflection API and a custom class loader;

28 Figure 3 is a flow chart illustrating an exemplary algorithm to be performed by a
29 custom class loader to implement the exemplary method for localization of Figure 2;

30 Figure 4 is a flow chart illustrating an exemplary algorithm to be performed by a
31 launcher to implement the exemplary method for localization of Figure 2; and

1 Figure 5 illustrates exemplary hardware components of a computer that may be
2 used in connection with the method for localization using a reflection API and a custom
3 class loader.

4 **Detailed Description**

5 Many applications need to provide multi-language support in the user interfaces.
6 For example, an application may support English and Japanese. A user may choose the
7 language during an installation or when the application starts. In some cases an
8 application determines the language based on the operating system's default language.
9 After the user specifies the language the application will communicate with the user in the
10 specified language. For example, if the user selects "Japanese", the application will
11 display its messages in Japanese.

12 The user interface of the application typically contains characters, words, phrases
13 and sentences in the user's language. These characters, words, phrases and sentences are
14 usually defined as "strings" in the application. These strings can be located in the
15 application code and/or in resource files transferred together with the application.

16 A method and corresponding apparatus for localization of a Java application using
17 a reflection API and a custom class loader use specifics of Java language to provide
18 localization of certain data elements, i.e., variables, of the application during class
19 loading. The method and apparatus for localization provide a technique to localize static
20 class variables, so that the data elements of a particular class are localized when the class
21 loader loads that class. The method and apparatus typically apply to localization of user
22 interface data elements defined as text.

23 The method and apparatus for localization reduce complexity of the Java
24 application by eliminating the function code dedicated to localization. The custom class
25 loader is added to the application to support localization by inspecting the application
26 code loaded from a storage device, such as a disk or Network, together with the
27 application. The custom class loader may be added during code transferring process to
28 inspect the code and convert all strings of the application to localized strings. As a result,
29 the application may become less complicated, because many programmatic errors may be
30 eliminated during the process.

31 In addition, the method and apparatus for localization increase productivity of
32 engineers who write language-independent code, and reduce memory consumption of
33 classes that use localized strings. Since all localization is accomplished when the class is
34 loaded and since the code that uses localization is independent of the code performing

1 localization, the method and apparatus for localization ensure better performance of the
2 application and afford better code reusability.

3 Figure 1 illustrates a standard class loading process. Class files 115 are loaded
4 from storage devices 110, such as disk or Network, to a bootstrap class loader 120. The
5 bootstrap class loader 120 then converts the class files 115 into Java classes 125 to be
6 inputted in a Java virtual machine 130. A class loader is an object that is responsible for
7 loading classes. Given a name of a class, the class loader may attempt to locate or
8 generate data that constitutes a definition for the class. A typical strategy is to transform
9 the name of the class into a file name and then read a “class file” of the name from the a
10 file system. Applications typically implement subclasses of `ClassLoader` in order to
11 extend the manner in which the Java virtual machine 130 dynamically loads classes. The
12 `ClassLoader` class uses a delegation model to search for classes and resources. Each
13 instance of the `ClassLoader` class has an associated parent class loader. When asked
14 to find a class or resource, the `ClassLoader` instance may delegate the search for the
15 class or resource to its parent class loader before attempting to find the class or resource
16 itself. The Java virtual machine’s 130 built-in class loader, i.e., the bootstrap class loader
17 120, does not have a parent class loader, but may serve as the parent class loader of the
18 `ClassLoader` instance. The Java virtual machine 130 typically loads classes from a
19 local file system in a platform-dependent manner. For example, on UNIX systems, the
20 virtual machine loads classes from a directory defined by a `classpath` environment
21 variable.

22 Figure 2 illustrates a custom class loading process that implements an exemplary
23 method for localization using a reflection API and a custom class loader. Compared with
24 the standard class loading process shown in Figure 1, a custom class loader 210 is
25 developed during class loading to load the Java classes 125 from the bootstrap class
26 loader 120. In a Java application, the custom class loader class 210 is typically a subclass
27 of `java.lang.ClassLoader`. The custom class loader 210 also retrieves resource
28 files 215 (described later) from the storage devices 110, and inspects the application code
29 in the resource files 215. The custom class loader 210, together with a reflection API 230
30 and a launcher 220, then converts all strings of the application to localized strings.
31 Finally, the custom class loader 210 passes localized Java classes 225 to the Java virtual
32 machine 130. The reflection API 230 is a part of standard Java library that obtains
33 descriptors of class variables. The launcher 220 typically creates the custom class loader

1 210, loads a startup class 240 of an application using the custom class loader 210, and
2 runs the application by invoking a main method of the application startup class 240. The
3 application startup class 240 is a class that starts the application, whereas the main
4 method is a method that initiates the application's functionality. The launcher 220 and
5 the custom class loader 210 classes are loaded by the bootstrap class loader 120. The
6 launcher 220 may need to use the reflection API 230 to invoke the main method of the
7 application startup class 240. The launcher 220 may need to have a static method `main`
8 `()`, and be specified as a main class of the application, by, for example, including the
9 launcher name as a parameter of a Java command. The Java virtual machine 130 may
10 call the method `main ()` to invoke the main method of the application startup class 240.

11 The following rules are defined for an exemplary implementation of the method
12 and apparatus for localization. Rule 1 is for distinguishing a plurality of localizable
13 variables from all other variables by a variable name, whereas Rule 2 is for locating a
14 resource string in the resource files 215 by a class name and the variable name. A
15 localizable variable is a class variable that, for example, belongs to a public class, is a
16 public static but not final variable, and has type `java.lang.String`. Rules 1 and 2
17 may be documented so that application developers can use the rules for defining the
18 localizable variables and specifying keys in the resource files 215.

19 Figure 3 is a flow chart illustrating an exemplary algorithm to be performed by a
20 load class method of the custom class loader 210 to implement the exemplary method for
21 localization. Referring to Figure 3, the custom class loader 210 first loads the resource
22 files 215 from the storage device 110 (block 305), and locates all localizable variables of
23 a class using the reflection API 230 and Rule 1 (block 310). A skilled Java programmer
24 may be able to implement this step given a definition of Rule 1. Then, for each
25 localizable variable, the custom class loader 210 finds a corresponding localized resource
26 file 215 for a current language according to Rule 2 by, for example, searching variables
27 with predefined prefix or postfix with `LOC_` (block 320). All values in the resource files
28 215 may need to be specified before deploying the application.

29 Next, the custom class loader 210 calculates a key for each localizable variable
30 according to Rule 2 by, for example, deleting the prefix and combining the class name
31 and variable name (block 330). Then, the custom class loader 210 finds a localized string
32 in the resource file 215 corresponding to each key (block 340). This step may be easily
33 implemented by a skilled Java programmer. Finally, the custom class loader 210 assigns

1 the localized string to the corresponding localizable variable, if the string is found (block
2 350). If more localizable variables exist (block 360), blocks 320-350 repeat.

3 Another embodiment of the method for localization generates resource files 215
4 for a default language (for example, English) using the same principles as the steps
5 described above. The resource files 215 may serve as documentation for developers that
6 create resource files 215 for other languages. This process utilizes initial values of the
7 localizable variables. The following Listing 1 shows a class with localizable variables
8 that have default values.

9 **Listing 1**

```
10 package com.hp.sgmgr.tree;  
11 public class Strings {  
12     public static String LOC_OBJECT_MANAGER = "Object  
13 Manager";  
14     public static String LOC_CLUSTERS = "All Clusters";  
15 }
```

16 In this embodiment, following block 340 of the algorithm in Figure 3, if the key
17 exists in the resource files 215, the process goes to block 350 (block 342). If not, the
18 custom class loader 210 appends the key and the value of the localizable variable to the
19 resource file 215 (block 344). This process may be invoked once after all localizable
20 variables are defined in the application.

21 After the custom class loader 210 is implemented, the launcher 220 may be
22 created with a method, such as a main method of the launcher 220, i.e.,
23 main(String[]). The launcher 220 loads and runs the application startup class 240
24 using the custom class loader 210. The launcher 220 typically calls only one method of
25 the application startup class 240, which runs the application. The name of the application
26 startup class 240, the name and signature of the main method of the application startup
27 class 240 may be application-specific. The launcher 220 may also pass parameters that
28 are passed to the launcher's main method to the application's main method.

29 Figure 4 is a flow chart illustrating an exemplary algorithm to be performed by the
30 launcher 220 (shown in Figure 2) to implement the exemplary method for localization.
31 Referring to Figure 4, after the launcher 220 is created, the launcher 220 first creates an
32 instance of the custom class loader 210 (block 410). Then, the launcher 220 requests and
33 loads the application startup class 240 from the custom class loader 210, by calling a

1 method, such as `Class.forName(String, ClassLoader)`, and specifies the
2 application startup class name as a first parameter and the custom class loader instance as
3 a second parameter (block 420). The custom class loader 210 localizes each class 240
4 loaded (block 442), and the method returns a localized version of the application startup
5 class 240 to the launcher 220. The custom class loader 210 may also load all classes
6 referenced by the application startup class 240, since by default a class is loaded by the
7 same class loader as the referencing class. For example, if class A refers to class B, and
8 class B is not loaded yet, then the Java virtual machine 130 uses the same class loader that
9 was used for loading class A to load class B. The launcher 220 and the custom class
10 loader 210 may not refer to any other application classes. Finally, the launcher 220
11 invokes the main method and runs the application startup class 240 using the reflection
12 API 230 (block 430).

13 All classes of the application loaded by the custom class loader 210 may access
14 the localizable variables, which may contain strings in the target human language at run
15 time. As a result, the method and apparatus for localization allows a Java application to
16 use a single human language during execution. The target language can be specified
17 either to the launcher 220 in the command line, or determined by the custom class loader
18 210 based on the default environment setting, or by any another method that is convenient
19 for the implementation.

20 The following illustrates an exemplary implementation of the method for
21 localization using a reflection API and a custom class loader. In this example, the rules
22 are identified as follows:

23 Rule 1: the variable contains a localized string if and only if its name begins with
24 “`LOC_`”.

25 Rule 2: the base name for the resource file is always
26 “`localize.properties`”.

27 The resource file name for the target language is constructed by passing the base
28 name and Java Locale object to `java.util.ResourceBundle` class. The
29 `ResourceBundle` class adds a postfix to the file name for locating the resource file
30 215 for specified human language. The resource file 215 is located in the same directory
31 as the class file containing the variable. A key is constructed by concatenating the class
32 name with variable name minus “`LOC_`” prefix, adding a dot (“.”) between the class name
33 and variable name.

1 Listing 2 shows an example of a class with localizable variables.

2 **Listing 2**

```
3 package com.hp.sgmgr.tree;
4 public class Strings {
5     public static String LOC_OBJECT_MANAGER;
6     public static String LOC_CLUSTERS;
7 }
```

8 According to Rule 1, the class in Listing 2 contains two variables:

9 "LOC_OBJECT_MANAGER" and "LOC_CLUSTERS".

10 According to Rule 2, the base path/file name of resource file 215 corresponding to
11 both variables in this class is "com/hp/sgmgr/tree/localize.properties".

12 The keys corresponding to these variables are "Strings.OBJECT_MANAGER" and
13 "Strings.CLUSTERS".

14 Listing 3 shows the resource file containing the values of localizable variables for
15 the class from Listing 2 for English language.

16 **Listing 3**

```
17 Strings.OBJECT_MANAGER=Object Manager
18 Strings.CLUSTERS>All Clusters
```

19 Referring to Listing 3, class ResourceBundle, which is used in the exemplary
20 implementation, supports this format of resource files 215. In this format, each line in the
21 resource files 215 contains one key-value pair, where both the key and value are strings,
22 and are divided by the "=" character.

23 In this example, the implementation of the custom class loader 210 uses class
24 ResourceBundle for retrieving localized strings by keys. The custom class loader
25 210 accepts the target language code as a parameter to its constructor, and passes the
26 target language code to ResourceBundle, which locates resource files 215
27 corresponding to the target language. The constructor is a method or procedure that is
28 invoked when an object is created in order to initialize the variables of the object. A class
29 may have more than one constructors, and the Java virtual machine 130 has rules that
30 define which constructor to call. Finally, the launcher 220 reads a configuration file
31 containing the target language specified at the time of installation, and passes the code of
32 target language to the custom class loader 210 to create the custom class loader 210.
33 Alternatively, the custom class loader 210 may also read the configuration file. As a

1 further alternative, the target language is passed as a parameter to the launcher 220 by a
2 user.

3 Figure 5 illustrates exemplary hardware components of a computer 500 that may
4 be used in connection with the method for managing data from multiple data sources
5 using conduits. The computer 500 includes a connection with a network 518 such as the
6 Internet or other type of computer or telephone networks. The computer 500 typically
7 includes a memory 502, a secondary storage device 512, a processor 514, an input device
8 516, a display device 510, and an output device 508.

9 The memory 502 may include random access memory (RAM) or similar types of
10 memory. The secondary storage device 512 may include a hard disk drive, floppy disk
11 drive, CD-ROM drive, or other types of non-volatile data storage, and may correspond
12 with various databases or other resources. The processor 514 may execute information
13 stored in the memory 502, the secondary storage 512, or received from the Internet or
14 other network 518. The input device 516 may include any device for entering data into
15 the computer 500, such as a keyboard, keypad, cursor-control device, touch-screen
16 (possibly with a stylus), or microphone. The display device 510 may include any type of
17 device for presenting visual image, such as, for example, a computer monitor, flat-screen
18 display, or display panel. The output device 508 may include any type of device for
19 presenting data in hard copy format, such as a printer, and other types of output devices
20 including speakers or any device for providing data in audio form. The computer 500 can
21 possibly include multiple input devices, output devices, and display devices.

22 Although the computer 500 is depicted with various components, one skilled in
23 the art will appreciate that the computer 500 can contain additional or different
24 components. In addition, although aspects of an implementation consistent with the
25 present invention are described as being stored in memory, one skilled in the art will
26 appreciate that these aspects can also be stored on or read from other types of computer
27 program products or computer-readable media, such as secondary storage devices,
28 including hard disks, floppy disks, or CD-ROM; a carrier wave from the Internet or other
29 network; or other forms of RAM or ROM. The computer-readable media may include
30 instructions for controlling the computer 500 to perform a particular method.

31 While the method and apparatus for localization using a reflection API and a
32 custom class loader have been described in connection with an exemplary embodiment,
33 those skilled in the art will understand that many modifications in light of these teachings
34 are possible, and this application is intended to cover any variations thereof.